CIS 624 Programming Languages

Spring 2000

Final (100 points)

Name (please print): ....

Id: ....

1	Objects	20 points	
2	More Objects	15 points	
3	Variable Offsets	20 points	
4	Call-by-need	15 points	
5	Callcc and Control Operators	15 points	
6	Types	10 points	
7	Inlining Using β-reductions	10 points	
	Total	105 points	

1 Objects (20 points)

Given the following Java declarations:

```
class A {
    int i = 3;
    int m1 () { return i + m2(); }
    int m2 () { return i + m1(); }
}

class B extends A {
    int i = 30;
    int m1 () { return i + super.m1(); }
    int m2 () { return i + m3(); }
    int m3 () { return i; }
}

class C extends B {
    int i = 300;
    int m1 () { return i + super.m1(); }
    int m3 () { return i; }
}
```

Answer the following questions:

- Draw a diagram that shows how a realistic JVM would internally represent an instance of class `C`.

- Draw diagrams that show how a realistic JVM would internally represent the classes `A`, `B`, and `C`.

- Trace the main steps needed to execute the call `new C().m1()`. Be sure to tell me what the result of the call is!

2 More Objects (15 points)

This problem builds on the implementation of the OO interpreter done in Homework 7. The main features of this implementation are summarized on the next page.

We will extend the OO language to be interpreted with a new construct:

```
datatype exp = ...
  | Cast of string * exp
```

The informal semantics is as expected. A `Cast(s,e)` expression evaluates `e` to an object reference, then checks that the class of the object is a subclass the class `s`. If the check succeeds, the evaluation returns the object reference as its result; otherwise the evaluation raises an exception.

The main changes needed to support the new construct are the following:

- add a new clause to the definition of `evalexp` in the `MakeInterpreter` functor,
- add a `isSubclass` function to the `OO_INTERPRETER` interface,
- add an implementation of the `isSubclass` function in the `OO_Interpreter` structure.

Describe these changes in as much detail as possible. An ideal solution would give the entire code, possibly relying on some non-public functions implemented in the homework, as long a short description of these functions is given.

```
functor MakeInterpreter (OO_Interpreter : OO_INTERPRETER) : INTERPRETER =
struct
  (* Implementation of an interpreter parametrized by the OO constructs *)
  fun evalexp exp env = ...
end

structure OO_Interpreter : OO_INTERPRETER =
struct
  (* Implementation of the OO constructs *)
end

signature INTERPRETER = sig
  val eval : Absyn.prog -> unit
end

signature OO_INTERPRETER = sig
  type object
  type value = object Env_Value.value
  type env = object Env_Value.env

  val elaborateClassDecls : Absyn.class list -> unit
  val getClassName : object -> Absyn.classname
  val find_method_and_apply :
      (Absyn.exp -> env -> value) ->
      Absyn.methodname -> Absyn.classname -> object -> value list
      -> value
  val new_object : Absyn.classname -> object
end

signature ENV_VALUE = sig
  datatype 'a value =
      Num of int
    | Closure of Absyn.parametername list * Absyn.exp * 'a env
    | List of 'a value list
    | Void
    | ObjectRef of 'a
    | Classname of Absyn.classname
  and 'a env =
      EmptyEnv
    | ExtendList of ('a env) * (string list) * ('a value ref list)
    | ExtendArray of ('a env) * (string list) * ('a value ref array)

  val lookupL : 'a env -> string -> 'a value ref
  val lookup : 'a env -> string -> 'a value
end
```

3 Variable Offsets (20 points)

I have alluded several times in class to the fact that variable names are replaced by offsets during compilation.

Consider the following simple source language and the test expression:

```
structure Absyn =
struct
    datatype exp =
        Var of string
      | Num of int
      | Plus of exp list
      | Lambda of string list * exp
      | App of exp * exp list

val test =
  App(Lambda(["a","b","c"],
        App(Lambda(["a","b"],
              Plus([Var "c",
                    Plus([Var "a", Var "b"])])),
            [Var "b", Var "c"])),
      [Num 1, Num 2, Num 3]);
end
```

The evaluation of expressions in the source language usually requires an environment which associates every variable with its value. For example, at the point where we are ready to perform the additions in the `test` expression above, the environment looks like:

```
FRAME 1:   (a b c) --> (1 2 3)
FRAME 2:     (a b) --> (2 3)
```

Because of the *static scope* rules, the *shape* (but not the actual contents) of that environment can be completely determined without evaluating the expression. Indeed, examination of the test expression should reveal that the shape of the environment at the point where we are ready to perform the additions is:

```
FRAME 1:   (a b c) --> (? ? ?)
FRAME 2:     (a b) --> (? ?)
```

The first frame corresponds to the first `Lambda` and the second frame corresponds to the second `Lambda`. Given this information, we can pre-compute the following (zero-based) offsets for the arguments to the additions:

- The value of the first argument to `Plus` (the value of `Var "c"`) can be found by traversing the environment as follows. Starting from the most recent frame, skip one frame, and get the value at index 2 in the next frame. We can summarize this information as the pair $(1, 2)$.

- The values of the arguments to the inner `Plus` are found at offsets $(0, 0)$, and $(0, 1)$ in the environment respectively.

Given this idea, we can compile the source language to a language in which all variable names have been replaced by offsets. In addition, `Lambda` expressions only need the number of arguments but not their names. The syntax of such a target language is:

```
structure Target =
struct
    datatype exp =
        Var of int * int
      | Num of int
      | Plus of exp list
      | Lambda of int * exp
      | App of exp * exp list
end
```

The evaluation of expressions in the target language can proceed using the offsets instead of the names. Here is a complete interpreter for the target language:

```
structure Eval =
struct
  structure T = Target

  exception Error

  datatype value =
      Prim of int
    | Closure of int * T.exp * env
  withtype env = (value list) list

  fun extend env vs = vs :: env
  fun lookup env d p = List.nth(List.nth(env,d),p)

  fun eval (T.Var (d,p)) env = lookup env d p
    | eval (T.Num i) env = Prim i
    | eval (T.Plus es) env =
      let val [Prim v1, Prim v2] = map (fn e => eval e env) es
      in Prim (v1+v2)
      end
    | eval (T.Lambda(n,e)) env = Closure(n,e,env)
    | eval (T.App(e,es)) env =
      let val (Closure(n,body,cenv)) = eval e env
          val args = map (fn e => eval e env) es
      in if (length args = n)
         then eval body (extend cenv args)
         else raise Error
```

```
      end
end
```

Write the function `translate` that converts from source expressions to target expressions. The result of converting the `test` expression above is:

```
  T.App(T.Lambda(3,
          T.App(T.Lambda(2,
                  T.Plus[T.Var (1,2),
                         T.Plus [T.Var (0,0), T.Var (0,1)]]),
                [T.Var (0,1), T.Var (0,2)])),
        [T.Num 1, T.Num 2, T.Num 3])
```

4 Call-by-need (15 points)

Call-by-need is the default parameter-passing mechanism in languages like Haskell. We will only consider the following small subset:

```
structure Absyn =
struct
    datatype exp =
        Var of string
      | Num of int
      | Plus of exp list
      | If of exp * exp * exp
      | Lambda of string list * exp
      | App of exp * exp list
end
```

The evaluation of the language proceeds in a familiar way: we have a notion of environments, a notion of value, and an interpreter that maps abstract syntax expressions and environments to values. The fundamental change is that arguments to functions are *not* evaluated at the time when the function is called. Instead enough information is saved in the environment (essentially a closure but traditionally called a *thunk*). Then when the value of a variable is *needed* (hence the call-by-*need*), this information is used to calculate a value for the variable. So far, we have described call-by-name. What distinguishes call-by-need from call-by-name is that, once computed, the value of the variable is used to update the environment so that later accesses to the variable need not recompute it.

Here are the beginnings of a call-by-need interpreter that you should complete:

```
structure Eval =
struct
  structure A = Absyn

  exception Error

  datatype value =
      Prim of int
    | Closure of string list * A.exp * env
  and entry =
      Evaluated of value
    | Thunk of A.exp * env
  withtype env = (string * entry ref) list

  fun extend env xs rs = ListPair.zip(xs,rs) @ env

  fun lookup [] x = raise Error
    | lookup ((y,r)::env) x = if x=y then r else lookup env x
```

```
fun eval ...
```

5 Callcc and Control Operators (15 points)

Write an interpreter for the following language:

```
datatype exp =
    Var of string
  | Lambda of string * exp
  | App of exp * exp
  | Callcc of string * exp
  | Throw of exp * exp
```

The informal semantics is the following. The functional subset has the same semantics as ML (call-by-value with static scope). The evaluation of a `Callcc(s,e)` expression captures the current continuation, binds it to the variable `s`, and then evaluates the expression `e`. The evaluation of `Throw(e1,e2)` evaluates `e1` to a continuation (say k), then evaluates `e2` to a value (say v). Instead of returning v to the current continuation, it ignores the current continuation and uses k instead.

You may either use ML's `callcc` and `throw` or write your interpreter in CPS.

6 Types (10 points)

For each of the following ML expressions, either give its type, or explain the type error:

```
let fun f x = x+1
in f 3
end

------------------

let fun f x = (x+1) div x
    fun g (h,y) = h( f (y+y) )
in g(f,3)
end

------------------

fun help_ack f = fn n => if n=0
                         then f 1
                         else f ((help_ack f) (n-1))

fun ack n = if n=0
            then (fn y => y + 1)
            else help_ack (help_ack (n-1))

------------------

let exception Error
    fun f x = if x then 3 else raise Error
in (f false) + 4 + (f true) + 5 + (f false)
end

------------------

datatype intlist = Empty | Cons of int * intlist

let fun f c = c
in f Cons
end
```

7 Inlining Using β-reductions (10 points)

A β-reduction formalizes the process of symbolically executing a procedure call. For example if `f` has the declaration:

```
fun f x y = x (y , y+1)
```

then a call to `f` in another part of the program can be *inlined* during compilation. For example, a compiler can rewrite the call `f(g,4)` to `g(4,5)` by doing a β-reduction and then performing the addition of the two constants.

List at least three reasons that would prevent an optimizing compiler from aggressively performing β-reductions to try to eliminate all function calls.